IOP Kernel ERS

Network Systems Development

B O'C
S S
J L

LAP + kernel

serial driver

Resources

TYPE	ID	
SERD	60	serial port A
SERD	61	serial port B
iopc	127	LAP port B
iopc	128	LAP port A
iopc	1	SWIM IOP driver
iopc	0	serial IOP kernel

Introduction

The IOP can be thought of as an intelligent I/O device. Although this particular implementation deals mainly with the SCC there is no reason other devices could not be mapped into the IOP 65C02 memory space. The implementation of the IOP Kernel was undertaken to accomplish several design goals.

- The Kernel should support devices other than the SCC.
- Provide arbitration for the clients using the IOP.
- Provide an environment for device drivers to operate without knowledge of how other devices are operating.
- Provide methods for installing and disabling drivers.
- The Kernel should remain simple.

A simple multi-tasking approach suggested by Ron Hochsprung is used to partition the IOP functions into three basic tasks.

- Kernel
- A Driver
- B Driver

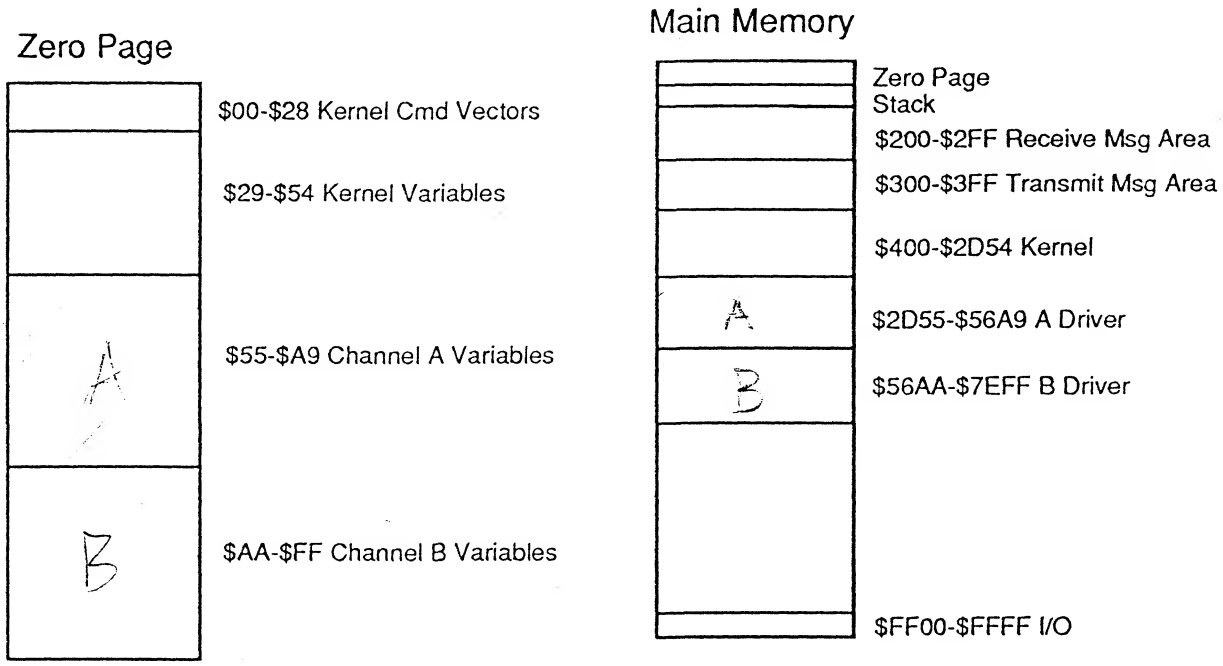
The Kernel task would operate continuously, installing Driver A or Driver B as task only when needed. In the SCC implementation the A and B Drivers would correspond to drivers used for the A and B channels of the SCC although other devices could be mapped into the SCC bit of the interrupt latch.

The Memory

The IOP is controlled by a 65C02 processor. Devices connected to the processor are generally memory mapped the SCC IOP being no exception. The memory map is an integral part of the overall design. The approach for the SCC implementation is to support only fixed memory partitions for each task. The reasons for this are:

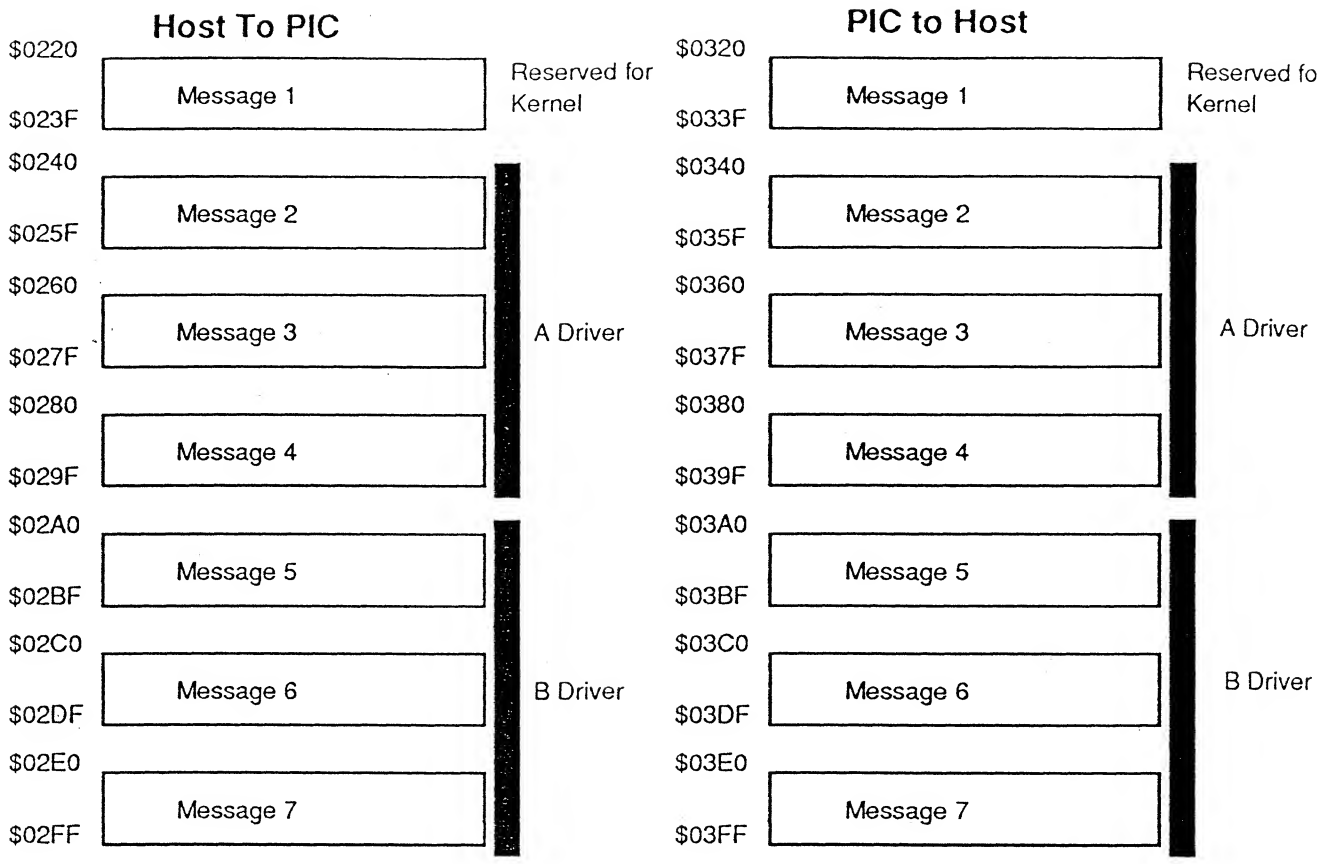
- Dynamic allocation would create significant over head.
- 65C02 code is hard to relocate.
- It would be difficult to stop, relocate and restart a task currently in operation.

Given the Kernel should eventually become stable and of fixed size the remaining memory will be divided equally between the two other task. Zero page allocation will be done the same. Zero page locations not used by the Kernel will be divided equally between the A and B drivers. Other special memory areas include the Host to IOP message area, the IOP to Host message area and the I/O memory map. The stack space for each task would also be divided equally.



Message Area

The message areas are fixed memory locations used to pass commands between the IOP and its host processor. There are seven message boxes in both directions each 32 bytes in length. Like memory the message boxes are divide equally among task. The first message box in each direction is reserved for the Kernel. The command format is only defined for the Kernel. Command formats for drivers will be left up to the driver writer.



Message ownership between the sending and receiving processors is controlled by 7 state bytes at the beginning of the respective message areas ($200 and $300). The command sender places the command in the appropriate message box, changes the message state to *NewMessageSent* and interrupts the receiving processor. Upon completion of the command the receiving processor fills the message with appropriate information, changes the state to *MessageComplete* and interrupts the sending processor to finish the transaction. All message passing is done via the IOP Manager described by Gary Davidian in the IOP ERS.

Kernel Commands

Currently the Kernel only receives commands from the host. An IOP to host message box was reserved for future use if needed. Kernel commands always begin with the command number followed by its parameters. Completed commands always return with the resulting error code followed by optional parameters. The Kernel supports six commands.

Allocate Driver Cmd # $01

```
struct    AllocCmdType {
                byte cmdNum;   /*  cmd # = $01      */
                byte Driver;   /*  Driver A =$00    */
                               /*  Driver B = $01   */
                byte ClientID ;  /*  Client ID > $00  */
              }
```

Allocate Driver is used to allocate driver memory within the IOP. A driver must be allocated before the actual driver code is down loaded into the IOP memory. The Allocate driver command begins with its message number ($01) and is followed by the *Driver* and *Client* parameters

Driver

The *Driver* parameter designates which driver is to be installed.

ClientID

The *ClientID* parameter is an identification byte assigned by the client attempting to allocate the driver. The ID byte of the client currently using the channel will be returned with the error *DvrInUse* when ever Allocate Driver is used on a driver that is already allocated.

Allocate Driver Error Messages

```
struct    AllocErrType {
                byte errNum ;      /* NoErr = $00
                                   Error  =  -1
                                   DvrInUse  = -4
                                   InByPass   = -5
                                 */
              byte ClientID; /* ID > 0 if driver in use  */
                             /*   or in ByPass          */
              }
```

NoErr

The driver has been allocated. The driver can now be down loaded.

Error

An invalid driver number was used.

DvrInUse

An attempt was made to allocate a driver that was already in use. The *ClientID* of the allocated client will be returned with this error.

InByPass

An attempt was made to allocate a driver while the IOP was in ByPass mode. The *ClientID* of the client that put the IOP in ByPass mode will be returned with this error.

DeAllocate Driver Cmd # $02

```
struct   DeAllocCmdType   {
                          byte cmdNum ; /* msg # = $02   */
                          byte Driver;  /* Driver A = $00*/
                                        /* Driver B = $01*/
                          }
```

DeAllocate Driver releases a driver for use by other clients.

Driver

The *Driver* parameter designates which driver is to be deinstalled.

DeAllocate Driver Error Messages

```
struct   DeAllocErrType  {
                           byte  errNum; /* NoErr = $00
                                            Error  = -1
                                            InByPass = -5
                                         */
                           byte ClientID; /* ID > 0 if in ByPass
                                             mode
                                         */
                         }
```

NoErr

Driver is deallocated no errors.

Error

An invalid driver number was used.

InByPass

An attempt was made to deallocate a driver while the IOP was in ByPass mode. The *ClientID* of the client that put the IOP in ByPass mode will be returned with this error.

Initialize Driver Cmd # $03

```
struct  InitCmdType {
                        byte CmdNum; /* cmd # = $03      */
                        byte Driver ;  /* Driver A = $00
                                          Driver B  = $01
                                       */
                    }
```

The Initialize Driver command is used after the driver has been down loaded into IOP memory. Each driver must have a Jmp InitProc located at the first address in driver memory. During initialization the Kernel will install this address as a task and wait for an InitFin event to occur. It is the InitProc's responsibility to signal the Kernel that the initialization has finished. The Initialize Driver message will then complete signifying the complete installation of the driver.

Driver

The *Driver* parameter designates which driver is to be initialized.

Initialize Driver Error Messages

```
struct  InitErrMsg    {
                        byte  errNum;  /* NoErr = $00
                                          Error  = -1
                                          InByPass  = -5
                                          NotAlloc  = -6
                                       */
                       byte ClientID; /* ID > 0 if in ByPass */
```

```
}
```

NoErr

Initialization completed no errors.

Error

An invalid driver number was used.

InByPass - An attempt was made to initialize a driver while the IOP was in ByPass mode. The *ClientID* of the client that put the IOP in ByPass mode will be returned with this error.

NotAlloc

An attempt was made to initialize a driver that is not allocated.

ByPass Mode Cmd # $04

```
struct   ByPassCmdType {
                        byte CmdNum;    /* cmd # = $04   */
                        byte On_Off;    /* ByPass Off = $00
                                           ByPass On  = $01
                                        */
                        byte ClientID;  /* ID > 0 of client
                                           who turns ByPass
                                           On.
                                        */
                    }
```

The ByPass Mode command controls whether ByPass Mode is On or Off. A client wishing to place the IOP in ByPass Mode must supply a *ClientID* byte identifying it to other clients trying to make Allocate and ByPass Mode calls.

On_Off

The On_Off parameter indicates whether the client wishes to turn ByPass on or off. Turning ByPass On maps the SCC directly into host memory. Turning ByPass Off maps the SCC into IOP memory. *The Kernel only executes the ByPass command when ByPass is On. All other messages will return InByPass error.*

ClientID

The *ClientID* parameter is an identification byte assigned by the client attempting to put the IOP in ByPass. The ID byte will identify the client to other clients that attempt to use the IOP while in ByPass.

ByPass Mode Message Errors

```
struct   ByPassErrType   {
                              byte errNum;     /* NoErr = $00
                                                  DvrInUse = -4
                                                  InByPass = -5
                                                  BadID = -7
                                               */
                            union {
                                  byte ByPassID; /* ID > 0 if in
                                                    ByPass
                                                 */
                                  byte DvrAID;    /* ID > 0 if
                                                     allocated
                                                  */
                                  } switch;
                            byteDvrBID;    /* ID > 0 if allocated */
                          }
```

NoErr

ByPass completed no errors.

DvrInUse

One or both of the drivers are allocated preventing the completion of the ByPass Mode command. DvrAID and DvrBID become valid and reflect the respective ClientID's of driver A and B.

InByPass

The IOP is already in ByPass mode. ByPassID is valid and identifies the client that put the IOP in ByPass.

BadID

An attempt to turn ByPass Mode Off was made with an ID other than the one used to turn ByPass On.

Version Request Cmd # $05

```
struct   VerCmdType {
                    byte  CmdNum;   /* cmd # = $05      */
                    byte  Driver;    /* Driver A = $00
                                        Driver B  = $01
                                        Kernel    = $02
                                    */
                 }
```

Each driver can register version information with the Kernel. The Version Request command will always return NoErr accept when the IOP is in ByPass Mode. The second parameter is a pointer into IOP memory where the version information is located. If the driver has not registered any version information the pointer will be Nil. The format for version information has not been decided.

Driver

The *Driver* parameter designates which driver to return version information.

Version Request Error Messages

```
struct   VerErrMsg   {
                    byte  errNum; /* NoErr = $00          */
                    ptr   VerInfo; /* VerInfo ≠ Nil if Driver
                                      has registered version
                                      info with Kernel.
                                  */
                 }
```

NoErr

Version information valid completed no errors.

The Kernel

The Kernel was designed to take advantage of 65C02 architecture.

The most powerful aspect of the 65C02 is the indirect and XY indexed addressing modes therefore many Kernel functions use table lookups for speed. The basic function of the Kernel is to provide Interrupt Dispatching, Multi-Tasking Support and Command Dispatching for the drivers. Intertask communication and interrupt event signalling is achieved through signals that can be sent to particular task.

Multi-Tasking

All tasking done by the IOP Kernel is purely voluntary. Tasks can relinquish control to wait for events or to allow other tasks to execute resuming after all the tasks have cycled through. The Kernel task is always installed and is constantly waiting for a NewCommandReceived event to occur.

Task context information is retained in the Task Control Block (TCB) data structure. There is one TCB allocated to each task. Each task is assigned a task ID. The task ID is always some multiple of the TCB size because the ID is used as an offset into the TCB data area. The Kernel ID is always $00 because it is the first task installed and uses the first TCB in the TCB data area. Driver A task ID is TCBSize and driver B task ID is 2*TCBSize.

```
TCB         Record

tPrev       DCB  Kern_ID      ; Previous Task
tNext       DCB  Kern_ID      ; Next task ID (initially 0 for
                              ; Kernel Task only)
tSp         DCB  $FF        ; Stack Pointer for this task
tEvent      DCB  $80        ; Event flags
                            ;  (bit 7 event always active)
tWait       DCB  $00        ; Event Mask

            EndRecord
```

The task queue is a forward and backward circularly linked list. The links are provided by the tPrev and tNext fields of the TCB. Links are not full addresses but are simply the task ID's bytes. When the Kernel is first installed as a task it is linked to itself.

The tSp field provides a location to save task stack pointers when tasks are switched. This allows each task to have it's own stack

area. No other state information is retained therefore tasks wishing to save other register information must do so themselves before releasing to the task event loop.

The task event loop is simply a tight code loop that scans the task queue for task whose events have occurred. The tEvent and tWait bytes are used by the task event loop. The task event loop logically AND's tEvent and tWait to determine if an event has occurred and the task should resume execution. If the result is zero the task event loop proceeds to the next TCB in the queue. A nonzero result causes the stack pointer to be changed to the tSp value, the Current Task variable maintained by the Kernel to be changed to the task ID of the TCB and resumption of the task execution.

The most significant bit of tEvent is always set. The Always Event should never be used as a signal bit or reset by any task. The Always Event allows the Release Task function of the Kernel to work properly. When a task wishes to release itself and allow other task to operate for a while it calls Release Task which sets the high bit in tWait of it's TCB. Because the high bit of tEvent is always set the task can be assured that execution will resume on the next task cycle.

Message Dispatching

The Kernel dispatches Host interrupts to the Kernel Message Scanner. The Kernel Message Scanner begins by disabling Host interrupts and enabling interrupts of higher priority. It then starts scanning the Transmit Message state bytes for MessageComplete. When a message is found to have a state of MessageComplete the Message Scanner dispatches to that messages Transmit Completion Signaller. The Transmit Completion Signaller would then send the task that is using the message box a MessageComplete Signal and return to the Message Scanner. After all of the Transmit state bytes are processed the Scanner proceeds to process the Receive Message state bytes by scanning for NewMessageSent and a similar process is repeated.

Transmit and Receive Message Signallers are installed by each driver for each message box that it intends to use. The Kernel installs its own message signaller for message box one. A message signaller should be short so that it does not hold up execution of the Message Scanner. This is the Kernel message signaller.

```
RxMsg_Signal Proc

        Import  Cmd_JmpTable   : Code

        Ldx  #Kern_ID          ; ID of task we wish to signal
        Lda  #RxMsg            ; Signal we want to send
        Jmp  Cmd_JmpTable+(SignalTask*3) ; Signal Task Cmd
                                         ;   has RTS

             endproc
```

When all the state bytes have been processed the Scanner enables host interrupts and returns.

Kernel Commands

Kernel command subroutines are available to all tasks. The command subroutine vector table starts a $0000 and extends to $0028, allowing for 20 command vectors. Currently there are only nine implemented. All parameters are passed to subroutines in the A, X and Y registers. Some subroutines use the carry to distinguish between complementary calls such as Install Receive Message Signaller and Remove Receive Message Signaller.

Because the 65C02 does not have an jump subroutine indirect address instruction the macros simulate one by using a jump indirect address table. Each macro assumes that there will be a label called Cmd_JmpTable available to it. The following table would be located at that label.

```
Cmd_JmpTable  Jmp    (Krn_CmdBase+(RemvRxMsg*2))
              Jmp    (Krn_CmdBase+(RemvTxCmpl*2))
              Jmp    (Krn_CmdBase+(RemvISR*2))
              Jmp    (Krn_CmdBase+(RemvSCCISR*2))
              Jmp    (Krn_CmdBase+(InstTask*2))
              Jmp    (Krn_CmdBase+(RelTask*2))
              Jmp    (Krn_CmdBase+(WaitEvent*2))
              Jmp    (Krn_CmdBase+(SignalTask*2))
              Jmp    (Krn_CmdBase+(ResetEvent*2))
              Jmp    (Krn_CmdBase+(ResetChan*2))
              Jmp    (Krn_CmdBase+(GetTMPB*2))
```

```
Jmp     (Krn_CmdBase+(InstTmTask*2))
Jmp     (Krn_CmdBase+(RegVer*2))
```

All global equates and macros are found in the include file "PICSys.i." Any Proc using a macro would have to import this table. The macros consist of code that loads the registers appropriately, sets or clears the carry and JSR's to the proper entry in the Cmd_JmpTable. Each driver would add this table to it's own code.

Install Receive Message Signaller

Parameters

X Register - message box number * 2
Y Register - High byte of signaller address
A Register - Low byte of signaller address

Carry Flag - Set

Macro Name - _Inst_RxMsgSgn MessageNumber, SignallerAddress

Install Receive Message Signaller places the address passed in A and Y registers in the Kernel's receive message signaller dispatch table. The message number times 2 contained in register X is the word offset into the table. There is one entry per message. The carry must be set for this call because Remove Receive Message Signaller uses the same command vector.

Remove Receive Message Signaller

Parameters

X Register - message number * 2

Carry Flag - Clear

Macro Name - _Remv_RxMsgSgn MessageNumber

Remove Receive Message Signaller places the address of an RTS in the Kernel's receive message signaller dispatch table. The message number times 2 contained in register X is the word offset into the table. There is one entry per message. The carry must be clear for this call because Install Receive Message Signaller uses the same

command vector.

Install Transmit Completion Signaller

Parameters

X Register - message number * 2
Y Register - High byte of signaller address
A Register - Low byte of signaller address

Carry Flag - Set

Macro Name - _Inst_TxCmplSgn MessageNumber, SignallerAddress

Install Transmit Completion Signaller places the address passed in A and Y registers in the Kernel's transmit completion signaller dispatch table. The message number times 2 contained in register X is the word offset into the table. There is one entry per message. The carry must be set for this call because Remove Transmit Completion Signaller uses the same command vector.

Remove Transmit Completion Signaller

Parameters

X Register - message number * 2

Carry Flag - Clear

Macro Name - _Remv_TxCmplSgn MessageNumber

Remove Transmit Completion Signaller places the address of an RTS in the Kernel's transmit completion signaller dispatch table. The message number times 2 contained in register X is the word offset into the table. There is one entry per message. The carry must be clear for this call because Install Transmit Completion Signaller uses the same command vector.

Install Interrupt Service Routine

Parameters

X Register - Interrupt type
Y Register - High byte of ISR address
A Register - Low byte of ISR address

Carry Flag - Set

Macro Name - _Inst_ISR InterruptType, ISRAddress

Install Interrupt Service Routine places the address passed in A and Y registers in the Kernel's interrupt dispatch table. The interrupt type contained in register X is the word offset into the table. There is one entry per interrupt bit in the interrupt latch. The carry must be set for this call because Remove Interrupt Service Routine uses the same command vector. The list of interrupt types is contained in "PICSys.i."

Remove Interrupt Service Routine

Parameters

X Register - Interrupt type

Carry Flag - Clear

Macro Name - _Remv_ISR InterruptType

Remove Interrupt Service Routine places the address of the UnKnown Interrupt handler in the Kernel's interrupt dispatch table. The interrupt type contained in register X is the word offset into the table. There is one entry per interrupt bit in the interrupt latch. The carry must be clear for this call because Install Interrupt Service Routine uses the same command vector.

Install SCC Interrupt Service Routine

Parameters

X Register - Interrupt type
Y Register - High byte of SCC ISR address
A Register - Low byte of SCC ISR address

Carry Flag - Set

Macro Name - _Inst_SCC_ISR InterruptType, SCCISRAddress

Install SCC Interrupt Service Routine places the address passed in A and Y registers in the Kernel's SCC interrupt dispatch table. The interrupt type contained in register X is the word offset into the table. There is one entry per interrupt type produced in the SCC read register 2. The carry must be set for this call because Remove SCC Interrupt Service Routine uses the same command vector. The list of SCC interrupt types is in "PICSys.i".

Remove SCC Interrupt Service Routine

Parameters

X Register - Interrupt type

Carry Flag - Clear

Macro Name - _Remv_SCC_ISR InterruptType

Remove SCC Interrupt Service Routine places the address of the UnKnown SCC Interrupt handler in the Kernel's SCC interrupt dispatch table. The interrupt type contained in register X is the word offset into the table. There is one entry per interrupt type produced by the SCC. The carry must be clear for this call because Install SCC Interrupt Service Routine uses the same command vector.

Install Task

Parameters

X Register - Task ID
Y Register - High byte of Task address
A Register - Low byte of Task address

Carry Flag - Set

Macro Name - _Inst_Task TaskID, TaskAddress

Install Task puts the Task Control Block (TCB) belonging to TaskID into the tasking queue. The Always event bit in the tEvent and tWait bytes of the TCB are set so that the task will automatically

execute when its cycle time arrives. The address contained in the A and Y registers is pushed onto TaskID's stack starting area. The stack (tSp) save byte for the TaskID's TCB is set to reflect that a address is on the stack. The carry must be set because Kill Task uses the same command vector.

Kill Task

Parameters

X Register - Task ID

Carry Flag - Clear

Macro Name - _Kill_Task TaskID

Kill Task removes the Task Control Block (TCB) belonging to TaskID from the tasking queue and returns to the caller. The carry flag must be clear because Install Task uses the same command vector.

Release Task

Parameters

None

Macro Name - _Release_Task

Release Task sets the high bit (Always Event) of tWait in the Task Control Block (TCB) of the current task that is executing then falls through into the task event loop. This allows a task to temporarily suspend execution resuming after one task cycle. Release Task clears any other bit that may have been set in tWait. The high bit of tWait is reserved for the Always Event Flag. The high bits in the tEvent bytes of all Task Control Blocks (TCB) are always set and should never be used as signal bits or reset using Reset Event.

Wait Event

Parameters

A Register - Event mask

Macro Name - _Wait_Event EventMask

Wait Event places the event mask in register A in the tWait byte of the current task's Task Control Block (TCB). Wait Event falls through into the task event loop suspending the current task until another task or event signals that the event or events have occurred. When event or events in Event Mask occur, register A will contain the results of the logical AND of tWait and tEvent when the task resumes execution.

The high bit of tWait is reserved for the Always Event Flag. The high bits in the tEvent bytes of all Task Control Blocks (TCB) are always set and should never be used as signal bits or reset using Reset Event. This bit is set in tWait by Release Task and insures the releasing task that it will resume on the next task cycle.

A Reset Event call should made before the next Wait Event is executed.

Signal Task

Parameters

X Register - Task ID
A Register - Signal Mask

Macro Name - _Signal_Task TaskID, SignalMask

Signal Task logically OR's the signal mask in register A with the tEvent byte in the Task Control Block (TCB) belonging to TaskID and returns to the caller.

Reset Event

Parameters

A Register - Reset Mask

Macro Name - _Reset_Event ResetMask

Reset Event logically AND's the reset mask in register A with the tEvent byte in the Task Control block belonging to the current task and returns to the caller. Reset Event should be called before the

next Wait Event call.

The macro Reset Event uses the complement of ResetMask.

Reset Channel

Parameters

Carry Flag - Set : Reset Channel A
Clear : Reset Channel B

Macro Name - _Reset_Chan Channel

A single register on the SCC resets both port A and B. Therefore it is necessary for the Kernel to arbitrait access to prevent corruption of other bits in the register. The macro is called with only one parameter.

```
_Reset_Chan A        ; Reset channel A
_Reset_Chan B        ; Reset channel B
```

Kernel Timer Task

The Kernel provides services for using the IOP's built in timer. Parameters are pasted to the Install Timer Task routine via the Timer Parameter Block. The Timer Parameter Block structure is:

```
struct  TMPBlk {
                byte RefNum;    /* Reference number      */
                Ptr  TimerTask; /* Address of timer task */
                word  TickCount; /* Timer tick count     */
              }
```

Memory for the TMPBlk is allocated by the driver. To start a timer task the driver must first get a Timer Reference Number for each task that is to be installed (current limit is 2 per driver). The driver then uses the Install Timer Task to start the Task. The Timer Task must be re-installed after each execution. If repeated execution is desired then the Timer Task should re-install itself with each execution. A Task can be cancelled by Cancel Timer Task. When a driver no longer needs a Timer Task the Timer Reference Number should be freed by Free Timer Reference Number.

Get Timer Reference Number

Parameters

None

Carry Flag - Clear

Returns

Y Register - Timer Reference Number

Macro Name - _Get_TMRefNum

Returns a Timer Reference Number in the Y Register. The RefNum is used internally by the Kernel to reference internal data structures related to the timer task.

Free Timer Reference Number

Parameters

Y Register - Timer Reference Number

Carry Flag - Set

Macro Name - _Free_TMRefNum RefNum

Return the Timer Reference number for use by the Kernel. The RefNum parameter of the macro is optional. If RefNum is omitted the macro assumes that the Y register already contains the RefNum.

Install Timer Task

Parameters

X Register - High byte of TMPBlk address
A Register - Low byte of TMPBlk address

Carry Flag - Clear

Macro Name - _Inst_TmTask Address

Install a Timer Task. The Address parameter is the address of the TMPBlk allocated by the driver.

Cancel Timer Task

Parameters

Y Register - Timer Reference Number

Carry Flag - Set

Macro Name - _Cancel_TmTask RefNum

Cancel a Timer Task. The RefNum parameter is optional. If omitted the macro assumes that the RefNum is already in the Y register.

Driver Format

The only restriction placed on driver formats is that is must be absolute code whose origin is located at the beginning of the Driver A or Driver B memory space. It must also contain initialization and close routines. The jump table for these routines should be located at the very beginning of the code.

```
StartDriver   Proc

              Jmp     InitProc
              Jmp     CloseProc
                 •
                 •
                 •
              EndProc

InitProc      Proc

              Install Message Handlers
              Install ISR's

              _Signal_Task  Kern_ID, IntFin   ; Signal Kernel Init

              Jmp Main
```

```
            EndProc
              •
              •
Main        _Wait_Event  MyEventMask
```

The Initialization code should install any message signallers and interrupt handlers needed for the driver. It should then signal the Kernel with the global equate InitFin indicating the initialization finished. The InitProc can then fall into its main event wait state.

Driver Initialization

The steps to down load and initialize drivers are as follows.

1. The client must first allocate the memory using the Allocate Driver message.

2. If the driver successfully allocates the driver code can then be down loaded into IOP memory.

3. Initialize the driver using the Initialize Driver message.

4. Begin communicating with driver.